Context-oriented Programming for Customizable SaaS Applications

Eddy Truyen[1], Nicolás Cardozo[2,3], Stefan Walraven[1], Jorge Vallejos[2], Engineer Bainomugisha[2], Sebastian Günther[2], Theo D'Hondt[2], Wouter Joosen[1]
eddy.truyen@cs.kuleuven.be, ncardozo@vub.ac.be

[1]IBBT-DistriNet, K.U.Leuven
Celestijnenlaan 200A - 3001 Leuven, Belgium
[2]Software Languages lab, Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels, Belgium
[3]ICTEAM, Université Catholique de Louvain
Place Sainte-Barbe 2 - 1348 Louvain-la-Neuve, Belgium

ABSTRACT

Software-as-a-Service (SaaS) applications are multi-tenant software applications that are delivered as highly configurable web services to individual customers, which are called tenants in this context. For reasons of complexity management and to lower maintenance cost, SaaS providers maintain and deploy a single version of the application code for all tenants. As a result, however, custom-made extensions for individual tenants cannot be efficiently integrated and managed. In this paper we show that by using a context-oriented programming model, cross-tier tenant-specific software variations can be easily integrated into the single-version application code base. Moreover, the selection of which variations to execute can be configured on a per tenant basis. Concretely, we provide a technical case study based on Google App Engine (GAE), a cloud platform for building multi-tenant web applications. We contribute by showing: (a) how ContextJ, a context-oriented programming (COP) language, can be used with GAE, (b) the increase in flexibility and customizability of tenant-specific software variations using ContextJ as compared to Google's dependency injection framework Guice, and (c) that the performance of using ContextJ is comparable to Guice. Based on these observations, we come to the conclusion that COP can be helpful for providing software variations in SaaS.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Extensibility*

General Terms

Design, Experimentation

Keywords

Multi-tenancy, Customization, Context-oriented programming, Software-as-a-Service, Google App Engine

1. INTRODUCTION

Software-as-a-Service (SaaS) [24] differs from traditional application service provisioning (ASP) in the sense that economies of scale play a much more important role. A traditional application service provider typically manages one dedicated server per customer. In contrast, SaaS providers typically adopt a *multi-tenant architecture* [5], meaning that a shared physical or virtual server typically hosts multiple customers, which are called tenants in this context.

Different architectural strategies can be applied to achieve multi-tenancy, for example at the application or virtualized infrastructure level. Most SaaS providers prefer the *application-level multi-tenancy* approach where multiple tenants are served by a shared application instance, which may be either placed on physical or virtualized hardware. The primary benefit of application-level multi-tenancy with respect to the virtualization approach is that the *operational costs can be significantly reduced* [5, 21]: (i) hardware and software resources can be more cost-efficiently divided and multiplexed across tenants, and (ii) the overall maintenance effort is seriously simplified because there is only a single version of the application code for all tenants.

A known problem with the application-level multi-tenancy approach however is the lack of customization flexibility. In order to meet the unique requirements of the different tenants, the application must be *highly configurable and customizable*. The current state of practice in application-level multi-tenancy is that configuration [5, 17] is preferred over customization which is considered too complex [23]. Configuration usually supports variance through setting predefined parameters for the data model, user interface and business rules of the application. Customization on the other hand involves software variations in the core of the SaaS application in order to address tenant-specific requirements that cannot be solved by means of configuration.

Existing programming models for customizable software such as reflection [4], aspect-oriented programming (AOP) [9] and dependency injection (DI) [10] do support tenant-specific variations to a certain extent, but these approaches also have several limitations with respect to management of tenant-specific customizations in a single-version application code base (a detailed account of these limitations is presented in Section 2). In this paper, we show that these limitations can be solved by the context-oriented programming (COP) model [8, 18].

Using COP, tenant-specific software variations can be easily integrated and managed in the core SaaS application. The selection of which variations to execute can be configured on a per tenant basis. Moreover, COP automatically ensures isolation between different tenants with respect to the application configuration, i.e. different combinations of software variations can co-exist within a single application instance at run-time. Because all tenants are served by the same instance of the SaaS application, COP supports run-time activation of tenant-specific software variations in such shared application instances.

This paper contributes with the following findings. First, we integrate the COP implementation ContextJ [18] on top of Google App Engine (GAE) [12]. While relying on the scalable and high-performant datastore of GAE to store and isolate tenant-specific application metadata, we use ContextJ to implement tenant-specific software variations. Second, we discuss how using ContextJ improves the flexibility and the customizability of software variations as compared to Google's dependency injection framework Guice [13]. Third, we show that the performance of using ContextJ is equal to Guice. Therefore, our approach is a viable alternative to current approaches of managing software variations in GAE.

The remainder of this paper is structured as follows. Section 2 gives an overview of the relevant state of the art in the field of tenant-specific customization. Subsequently, Section 3 introduces our application case study and sets out the requirements for tenant-specific customization of multi-tenant applications. Section 4 presents the COP programming model and its integration with Google App Engine. In Section 5 we illustrate the major benefits of COP with respect to customization flexibility and evaluate the performance overhead of COP+GAE with respect to operational costs by comparing it with the dependency injection approach Guice+GAE. Section 6 concludes the paper and discusses future work.

2. RELATED WORK

This section discusses related approaches and programming models with respect to tenant-specific customization of SaaS applications.

Virtualization-level Multi-tenancy

With virtualization-level multi-tenancy, virtualization technology can be used to run multiple operating system partitions with a dedicated application-middleware-OS stack for each tenant on shared servers. The advantage of this approach is its increased customization flexibility, i.e. it is possible to perform off-line customization of the application software for a specific tenant into a new version and then (re)deploy a virtual image with the new version for that tenant. Moreover the application design should not be tenant-aware and therefore the upfront application engineering cost is not increased [21]. On the negative side, however, server resources are unnecessarily wasted because for each tenant a separate instance of the middleware stack has to be created and maintained. Moreover, maintaining separate application code versions for each tenant also incurs a high maintenance cost which cannot be tolerated by many SaaS providers. For example, experiments by Tsai et al. [26] demonstrate that hosting multiple tenants on a single VM yields a scalability increase of 60-90% over a traditional design in which each tenant is assigned to a single virtual machine.

Multi-tenant Middleware

To manage multiple tenants on a single VM, an additional middleware layer is necessary to ensure appropriate data and performance isolation between different tenants. For example, Google App Engine (GAE) [12] recently introduced an API for isolation of application data of different tenants within a shared application instance [14]. Another initiative is a recent JSR to include support for multi-tenancy in the Java EE platform [7]. The aim of these middleware platforms is focused on reducing the application engineering complexity of building multi-tenant software by shifting this complexity from the application design towards a reusable middleware architecture. However, these approaches currently do not tackle the challenge of advanced customization, i.e. managing tenant-specific software variations within a single application instance. To support tenant-specific software variations, separate application instances still have to be created and maintained on a per tenant basis [2].

Programming Models for Customizable Software

Different software engineering models are used to encode software variations of a single core application. A meta-object protocol (MOP) allows programs to inspect (*reflection*) and modify (*intercession*) programs behavior. Using these reflective capabilities, it would be possible to differentiate, manage and modify an application for a particular tenant, much in the way it is done with classboxes [4]. However, reflection mechanisms require the host language to have a full MOP implementation, which is often not the case for industrial languages such as the ones used by SaaS providers.

Several aspect-oriented programming (AOP) [9] languages such as CaesarJ [1], allow dynamically scoped activation of program definitions which can be used for the introduction of tenant-specific behavior, and effectively express the conditions under which each aspect is applicable. Nonetheless, declaration of such conditions is performed statically, making it ill-suited for the maintenance of multiple tenants, since the configuration for each tenant will block the application for all the others.

Dependency injection (DI) is a well-known design pattern for component-based applications that enables to separate the management of component dependencies from the application code based on the principle of Inversion of Control (IoC) [10]. This pattern is supported by various dependency injection frameworks such as Guice [13] or IoC containers such as Spring [22]. In previous work, we have shown that it is possible to support tenant-specific software variations on top of a PaaS platform by means of dependency injection [27]. Still, there are some limitations inherently related to the component-based composition mechanism that de-

pendency injection assumes. Essentially, with dependency injection, a multi-tenant application is designed as a software product line with run-time binding of software variations [27]. However, software variations must be decomposed according to multiple localized variation points which are similar to hot spots in object-oriented framework design. Moreover, at most one software variation can be activated per variation point while in the general case it would be desirable to combine multiple ones.

3. MOTIVATION

This section motivates our work by means of an application case and subsequently sets out important characteristics for supporting tenant-specific customization of multi-tenant applications.

3.1 A SaaS Application for On-line Hotel Booking

Let us consider the example of a SaaS provider for on-line hotel booking (see Fig. 1). It provides a highly configurable service that travel agencies can use for booking hotels on behalf of their customers. In this scenario, travel agencies play the role of tenants whereas employees and customers of a travel agency correspond to a tenant's users. Employees are offered a customized user interface and customers of the travel agency can login to check the status of the travel items through a URL with a custom-made domain-name that corresponds with the travel agency. A special 'tenant administrator' role is assigned to someone who is responsible for configuring the SaaS application, setting up the application data and monitoring the overall service. This role can be played by an internal or external client of the SaaS provider. In the context of this simple example, the tenant administrator belongs to the ICT staff of a travel agency.

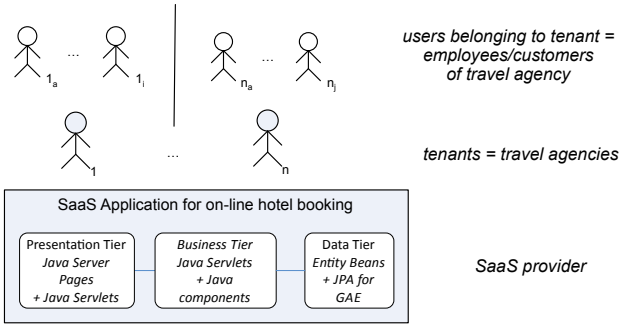

Figure 1: SaaS application for on-line hotel booking.

We designed this SaaS application as a multi-tiered Model-View-Controller web application and deployed it on top of Google App Engine (GAE) [12]. The View is implemented using Java Server Pages (JSP) at the presentation tier. The Controller is implemented using Java Servlets and plain old Java components; this controller logic is typically distributed among the presentation tier and business tier. Finally the Model is implemented in the data tier using JEE Entity Beans and a slightly modified version of the JPA standard [6, 3] as provided by the GAE Java SDK.

3.2 Tenant-specific Customization Requirements

We set out a list of important requirements for tenant-specific customization based on a simple customization scenario.

Take the case of a particular travel agency that wants to be able to offer discounts to their returning customers or during the low season. The hotel booking application should be extended with an additional service for managing customer profiles and a variation on the service for calculating prices. We assume furthermore that SaaS providers employ a business model where the base application is offered to tenants at no or low cost, but tenants incur an additional price for additional services. As stated above, we target the design of interactive web applications instead of coarse-grained service compositions or workflows. As a consequence, the implementation of these additional pricing and customer profile services are ideally integrated as part of the web application code base in the form of easy-to-manage software extensions in the above MVC-based architecture. Based on this simple customization scenario, business model and application design assumptions, we can derive requirements with respect to application development, application configuration and run-time support.

Tenant-specific software variations: The application development team of the SaaS provider should be offered a simple way to manage the different tenant-specific variations as separate units of deployment that can be selectively bound to the core architecture of the application.

Configuration facility: With respect to customization, tenant administrators should be offered a configuration facility to select what software variations should be enabled for them (e.g. the price calculation service). In addition, this facility should also allow to specify specific configuration parameters (e.g. business rules for the price calculation service). These configuration data should be stored in the datastore of the SaaS provider in an isolated way under a specific tenant ID.

Run-time activation of tenant-specific variations: Run-time support is needed to provide support for activating software variations on a per tenant basis or even per user basis. When a user (either client or employee) logs in, the tenant to which the user belongs should be determined. Based on the acquired tenant ID, the run-time support should then activate the appropriate software variations to process the requests of the user. Another key requirement of the run-time support is that the tenant-specific software variations should be applied in an isolated way without affecting the service behavior that is delivered to other tenants or users of that tenant. To ensure robustness in the face of run-time adaptations (i.e. safe updates [20]), we assume for now that multi-tenant applications implement the stateless session model and that all tenant-specific state is stored in a (separate) database.

4. SUPPORTING CO-EXISTING SOFTWARE VARIATIONS USING COP

This section presents how the context-oriented programming (COP) model supports tenant-specific customization

of SaaS applications. First, we briefly explain the supporting infrastructure of Google App Engine (GAE) that enables implementing multi-tenant applications without support for tenant-specific customization. Subsequently we introduce the COP programming model and explain its integration with GAE. To do so we use ContextJ [18], a context-oriented extension to the Java language.

4.1 Multi-tenancy Support by GAE

We propose to integrate COP as an extension to Google App Engine (GAE). The main reason is that this PaaS platform provides the supporting infrastructure to take care of scalability. Google App Engine automatically scales up (and down) by creating a pool of identical application instances as user load increases.

In our case, each application instance itself is able to serve multiple tenants. To achieve tenant data isolation within such a single application instance, three main components are required: (i) the *tenant context* containing the information of the tenant linked to the current request (via a unique tenant ID), (ii) *tenant-specific authentication* to identify the tenant, and (iii) *multi-tenant data storage*. As stated in Section 3, GAE has built-in support for such tenant data isolation via the Namespaces API [14]. We only had to implement a filter to map incoming requests to a specific tenant namespace. Moreover we have used the caching service of GAE [16] in order to retrieve tenant-specific data and metadata without large I/O performance overhead.

In order to enable tenant-specific customization, we have also implemented a simple configuration interface for the tenant administrators to express their preferred software variations. These tenant-specific configurations are stored as tenant metadata in the GAE datastore in the form of mappings from a Tenant ID to a set of layers. This concept of layers is the key of context-oriented programming and will be introduced in the next subsection.

4.2 Context-oriented Programming on GAE

To develop tenant-specific customizations, we use the runtime behavior adaptation facilities provided by an upraising programming paradigm, called *context-oriented programming* (COP) [18, 11, 8]. COP is especially tailored for the construction of highly adaptable software. It enables writing applications that can vary their behavior dynamically according to their *context* of execution. The context may include any computationally accessible information, ranging from the internal state of the application, e.g. user preferences and CPU usage, to external state such as the time of day or physical location. COP provides dedicated language-level abstractions to enable context-dependent software variations to be defined independently from each other and the core application. In addition, COP allows such variations to be dynamically *activated*, i.e. selected and composed with the core application.

Using COP, tenant-specific customizations can be expressed as independent software variations. These variations can be activated according to context information like the current tenant ID and the user role who interacts with the application. We illustrate the use of COP with the on-line hotel booking application introduced in Section 3. As shown in Figure 2, the application consists of three (vertical) tiers: the presentation tier, the business tier, and the data tier. We separate the behavior of the application into entities called *layers*. At the bottom, the *base* layer represents the core behavior of the application. In addition to this, we add the layers *LowSeason* and *VIP* which correspond to tenant-specific software variations. As can be seen, these layers modify the price calculation (e.g. providing low-season and VIP discounts) and the creation of bookings (e.g. allowing VIP users to create multiple tentative bookings).

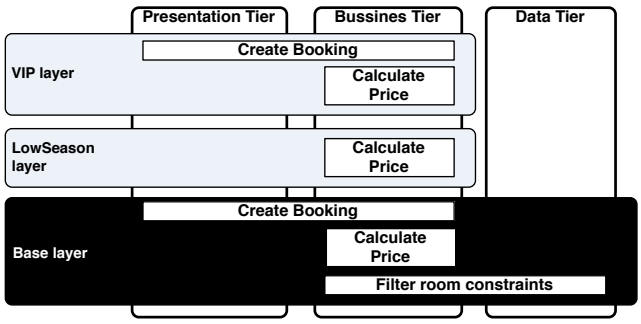

Figure 2: Software variations for the on-line booking application.

A *layer* is a first-class programming language entity in COP that represents and groups software variations. For the implementation of our case study, we use a Java framework for COP, called ContextJ [18][1]. As shown in Listing 1, layers are defined as separate objects.

```
1 public static final Layer VIP = new Layer();
2 public static final Layer LowSeason = new Layer();
```

Listing 1: Basic layer definition.

After creating the classes, we define and associate the software variations. In ContextJ, layer-specific behavior is defined by enclosing method declarations in `layer` expressions. Consider Listing 2, in which the `PriceCalculator` class is defined. This class provides a common method `calculateBookingPrice` that is part of the base layer (Lines 3–6). Then, we add specific method declarations for the `VIP` layer (Lines 8–14) and for the `LowSeason` layer (Lines 16–21).

Once the software variations have been defined, they are made available to the application by activating the corresponding layer. In ContextJ, a layer should be activated explicitly by means of the `with` construct. This construct contains a block-like set of expressions. These expressions are compiled with the software variations of the activated layer as their context. Outside of the `with` constructs, only the core behavior of the *base layer* is active.

Listing 3 shows a layer activation in ContextJ. Here, we see part of the definition of the `CreateBookingServlet` class. In Line 7, we use `getTenantLayers()` to retrieve the tenant-specific configuration and to determine the currently activated layer, for which we then execute the layer-specific `createBooking` method. For example, if the `VIP` layer would be

[1]The full implementation of ContextJ is available at [19]. ContextJ extends Java's standard syntax by means of a precompiler. There also exists an implemented version that does not use syntactic abstractions. We refer the interested reader to [18] for further details about these two implementations.

```
1 public class PriceCalculator {
2  ...
3  public double calculateBookingPrice(double price,
4      Date start, Date end) {
5    //Calculate default price
6  }
7
8  layer VIP {
9   public double calculateBookingPrice(
10      double price, Date start, Date end) {
11    double defaultPrice = proceed();
12    // Apply discount to default price
13  }
14 }
15
16 layer LowSeason {
17  public double calculateBookingPrice(
18      double price, Date start, Date end) {
19    // Calculate price with low season discount
20  }
21 }
22}
```

Listing 2: Software variation definition.

activated, we would use its software variation definition, and otherwise the behavior as it is defined in the *base layer*.

```
1 public class CreateBookingServlet
2                          extends HttpServlet {
3  ...
4  public void doPost(HttpServletRequest req,
5     HttpServletResponse resp) throws IOException {
6   ...
7   with(getTenantLayers()) {
8     // set booking constraints
9     createBooking(constrs, guest, priceCalc);
10    // createBooking calls calculateBookingPrice()
11  }
12
13  public Booking createBooking(Constraints constr,
14      User guest, PriceCalculator priceCalc) {
15    // Create single booking
16    // Display single booking
17  }
18
19  layer VIP {
20   public Booking createBooking(Constraints constr,
21       User guest, PriceCalculator priceCalc) {
22     // Create multiple bookings
23     // Display multiple bookings
24   }
25  }
26}
```

Listing 3: Software variation activation.

A tenant may be able to compose different layers, for instance to combine the behavior of a tenant-specific layer with the behavior of the *base layer*.[2] The call of a `proceed` construct in a method defined in the tenant-specific layer (e.g. in the `calculateBookingPrice` method in the `VIP` layer (Line 11 in Listing 2)), ensures that the original definition of the method is called.

The tenant-specific configurations used for choosing a specific layer in the `getTenantLayers()` method, are stored as tenant metadata in the GAE datastore in the form of mappings from Tenant ID to a set of layers. Similarly, metadata about user-specific software variations can also be stored in this way in the GAE datastore. In case that a specific tenant does not specify any preference, the behavior of the *base layer* is selected.

[2]In ContextJ, the order of composition of layers is determined by the order of activation. The base layer is considered to be activated before any other layer. We refer the reader to [18] for further details about layer composition in COP.

5. FEASIBILITY AND PERFORMANCE DISCUSSION

To show that ContextJ is a viable alternative to implement tenant-specific software variations in GAE, we implemented two versions of the on-line hotel booking case study, with Guice and ContextJ respectively. In this section we present and discuss our observations. First, we describe some details of the Guice implementation of the on-line booking application on top of GAE. Subsequently we describe the increased customization flexibility that context-oriented programming (ContextJ) gives over dependency injection (Guice). Finally, we evaluate and compare the performance of both implementations, and see that they have an equal performance.

5.1 The Guice-based Implementation

As explained in Section 2, Guice supports the binding of software variations to locally declared variation points. This binding is specified in a separate configuration file, called `web.xml`. To use Guice for supporting tenant-specific customizations on top of GAE, we first stored the data from this configuration file in the database as first-class entities that can be inspected and modified at run-time by tenant administrators. Second, to enable a single application instance to simultaneously serve multiple tenants, it should be possible to switch between software variations at run-time, i.e. run-time rewiring of dependencies must be implemented. Fortunately, Guice supports this run-time rewiring by injecting a `Provider` [15], which is a user-specified class to return an instance of a given type. In our case we implemented a `Provider` that returns for each variation point the software component that is specified in the configuration of the current tenant being served.

5.2 Customization Flexibility

Context-oriented programming succeeds to provide a flexible implementation of multi-tenant applications for GAE, with the added value that this can be done even at run-time. Under the spotlight of the case study presented in Section 3, the use of COP to customize SaaS applications is proven beneficial with respect to: modularization, configuration, and behavior isolation of software variations.

Tenant-specific software variations can be mapped to layers in a straightforward way. A system-wide customization, regarding a specific software variation, is grouped and managed together in a layer, which can be dynamically activated and deactivated. Dependency injection (DI) only offers support for localized changes. For example, the `VIP` layer in Figure 2 provides a software variation for creating a booking, which customizes the core behavior in both the business and presentation tiers.

COP supports the dynamic combination of multiple layers. For example, a specific tenant (i.e. travel agency) can configure the SaaS application to select both the `LowSeason` and `VIP` software variations using the configuration facility.

When a VIP customer of that tenant uses the on-line hotel booking application during the low season period, the two layers will be activated. To achieve layer combinations, it suffices to use the `with` construct at the entry point of the application, e.g. the `CreateBookingServlet` class in Listing 3. This kind of combinations are generally difficult to achieve using DI, where for each variation point it is possible to inject only one software component at a time.

Software variations are applied within the *dynamic scope* of the `with` construct. This property allows COP to inherently support the isolation of layer activations between different tenants. Different combinations of layers can thus be concurrently activated for different tenants. For instance, if a travel agency selects the `LowSeason` software variation, the appropriate layer will be activated only for the customers of that particular travel agency, while customers of tenants who have not selected this variation will not see the price discounts for the low season. With DI, a software variation is applied within the scope of a single variation point. There, the `Provider` ensures the isolation of this software variation.

5.3 Performance and Scalability Evaluation

For the performance evaluation, we first describe the cost model that we use for the execution cost of customizable SaaS applications. Next, we compare our COP implementation (using ContextJ), with the version developed using the Guice dependency injection framework. We describe the performed experiments on these two versions deployed on top of Google App Engine, and analyze the results.

5.3.1 Cost Model

With traditional application service provisioning (ASP), the execution cost is defined by the usage of CPU, memory and storage for normal application execution. In general, a customizable, multi-tenant SaaS application has the same application execution cost per tenant. However, performing tenant-specific authentication and the necessary customizations add to this base load. CPU time is increased to process the authentication and the customizations, and additional memory is needed to store the tenant-specific configurations, the different software variations, and data about the tenants (e.g. the tenant's name and address). Currently, storage is a cheap resource and has less influence on the total execution cost. Therefore, we focus our evaluation on CPU time and memory.

5.3.2 Experiments

We compare the execution costs, measured in CPU time and memory, of our case study for the Guice and the ContextJ version. Both versions are deployed with GAE (SDK 1.5.2), using the high replication datastore (default option). Each tenant is represented by 200 users who each execute a booking scenario. This booking scenario consists of 10 requests to the application: first several queries are performed to search for hotels with free rooms in a given period, then a tentative booking in one hotel is created, and finally the booking is confirmed. The different users of one tenant will execute the booking scenario sequentially, while the tenants run concurrently. It is not our goal to create a representative load for this application, but to compare the execution cost of the different versions under the same load. We retrieve the information about the execution cost via the Google App Engine Administration Console. It provides a dashboard displaying the resource usage by the application. Notice that the focus of this comparison is on the relative differences between the execution costs, since the absolute numbers depend on the current (global) load on the GAE platform. Furthermore, to keep the comparison between COP and DI fair, we only compared the performance of running the two applications with the same software variation (i.e. the `LowSeason` software variation) activated for all tenants on top of the base code. We also activated only this one software variation because Guice does not support combinations of multiple software variations at a single variation point.

In Figure 3 we present the evolution of the average CPU usage by the COP and DI versions with an increasing number of tenants. To our surprise, the COP version results in lesser CPU usage for a higher number of tenants.

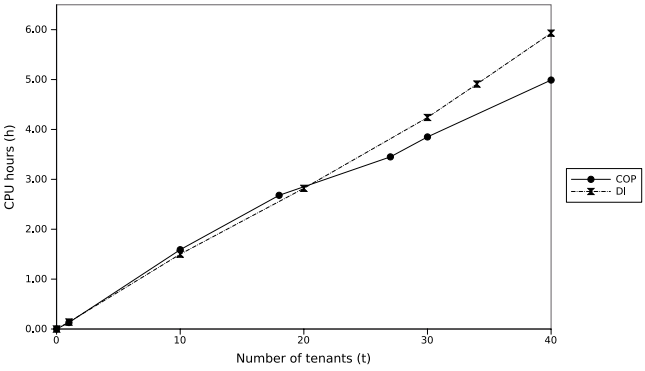

Figure 3: Overview of the CPU usage by the COP and DI versions.

The total memory usage cannot be measured precisely, because several other factors despite the application binaries add or reduce memory consumption: a rising number of requests triggers an increase in memory because a new instance is started to provide better load balancing, and once the requests decline, instances become idle and are removed to release memory. Therefore, we use the average number of instances to represent the maximal possible memory usage. Figure 4 shows the evolution of the average number of application instances when increasing the number of tenants. As can be seen, the number of instances increases slightly with the number of tenants, which is parallel for both versions.

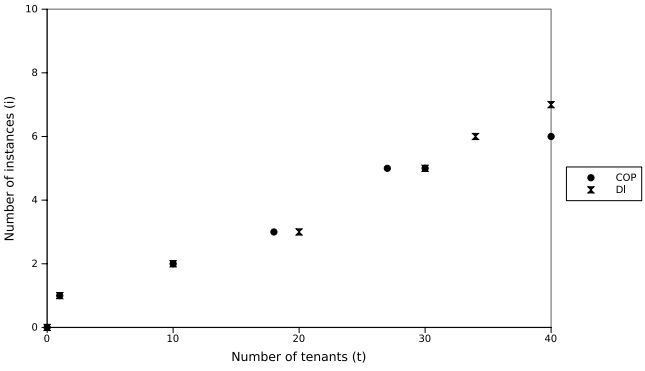

Figure 4: Overview of the number of instances used by the COP and DI versions.

Regarding scalability, Figure 4 also shows that GAE provides the same behavior of adding instances for both the Guice and the ContextJ versions, from which we conclude that COP does not have a negative effect on the scalability property of the application.

We also measured the number of datastore queries. The results of these measurements are almost identical for both implementations with minimal differences. As will be explained in the next subsection, the reason for this is that the `LowSeason` layer corresponds to a single variation point.

5.3.3 Analysis of the Results

The experiments clearly show that COP does not introduce any performance overhead compared to the implementation using dependency injection. Specifically, comparing the CPU load in Figure 3, we see that COP has a lower execution cost with an increasing number of tenants. The memory consumption, Figure 4, is equal for both versions. From these observations, we extrapolate that activating more than one software variation for each tenant will further increase the CPU load and memory consumption, while the scaling behavior remains the same.

A higher number of software variations will also cause a difference in the number of datastore queries between COP and dependency injection: we assume that the DI implementation increases the load more than the COP version when cross-tier software variations are considered. In the DI implementation, the appropriate tenant-specific configuration must be retrieved from the datastore at every variation point throughout the request handling process. Of course this will be optimized by caching, but even queries to the caching service cause I/O delays. In COP, the tenant-specific configuration for a particular request is retrieved only once from the datastore, that is, during the evaluation of the `with` construct. Thus, an increasing number of variation points has less impact on the COP version with respect to the number of datastore queries.

During the experiments, we encountered two challenges that are related to performing a big amount of requests and the state of sessions.

During the experiments, we experienced that GAE exposes an issue that user requests are dropped when the load is too high for a particular application instance to handle in time. This resulted in a temporary denial to handle user requests for all tenants that are sent to that instance, in particular those tenant applications for which new user requests arrive (i.e. at the start of a new tenant's client application). This unpredictable behavior especially appeared with a higher number of tenants. We decided to stop a tenant's client application from running when its user requests were dropped to prevent that other tenants would be affected. This explains why in Figure 4 the graphs of the different versions show results for a different number of tenants (for example 27 tenants for the COP version and 34 for the dependency injection version). In addition, this problem prevented us to perform experiments with a higher number of tenants.

As stated above, we assumed a stateless session model. However, even in a stateless session model, intermediate transient state is created in the course of a single transaction request. For example, in the on-line booking application, one or more bookings are first tentatively created during a session and the resulting state is stored in the caching service of GAE; thereafter during check-out all bookings are committed into permanent orders as a whole. Depending on the application design, this intermediate state needs to be managed in the scope of a single request, per session or simply per tenant. A powerful feature of dependency injection (and Guice in particular) is that these state scopes can be declaratively enforced and custom state scopes can be defined. COP on the other hand lacks such state scoping mechanism. This lack of COP does however not affect the robustness of the application because GAE's caching service actually implements the state scoping mechanisms automatically: transient state is managed in the caching service in such a way that the state cannot leak to other sessions or tenants. Still, an interesting line of work for COP specifically is adding a state scoping mechanism to COP.

6. CONCLUSION AND FUTURE WORK

The paper has shown that COP is a powerful customization mechanism that allows to easily integrate and manage tenant-specific software variations within SaaS applications. COP achieves a higher customization flexibility in comparison to the Guice dependency injection framework without causing any performance overhead.

We have experienced problems with COP in asynchronous communication primitives. When an application component, for example, spawns method invocations in a new thread, the layer activation logic of the originating thread should hand over to the new thread. This is currently not supported by the existing COP implementations. Existing middleware approaches that support context-sensitive customization tackle this problem by propagating activation logic with messages that are exchanged between processes [25]. Integrating such middleware solution with COP is an interesting avenue for future research.

Further, a big research challenge with application-level multi-tenancy is to create better support for ensuring performance isolation between co-existing tenant applications in a shared address space. During our experiments we experienced that GAE lacks performance isolation between the different tenants. This resulted in the denial to handle requests of certain tenants, especially when the number of tenants using the application increased. Moreover, besides this problem, it is desirable to assign different thread priorities to different tenants; however in a Java-based system additional support from the operating system layer is needed for this.

Finally, maintaining global state consistency comes at risk in the presence of run-time layer upgrades: safe state consistency must be enforced for those layers subject to upgrade, while on the other hand tenants that are not affected by the layer upgrade should not experience any interruption in the service delivery. This is also an important open issue.

7. ACKNOWLEDGMENTS

This work has been supported by the Research Foundation - Flanders (FWO) in the context of the RECOCO project, by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels, and by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy.

8. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[2] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA middleware for cloud computing. In *IEEE International Conference on Cloud Computing*, pages 458–465, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[3] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR '11: Proceedings on Conference on Innovative Data Systems Research*, 2011.

[4] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. *Modular Programming Languages*, pages 122–131, 2003.

[5] F. Chong and G. Carraro. Architecture strategies for catching the long tail. Microsoft Corporation, `http://msdn.microsoft.com/en-us/library/aa479069.aspx`, April 2006.

[6] L. DeMichiel. JSR 317: Java^{TM} Persistence 2.0. `http://www.jcp.org/en/jsr/detail?id=317`, 2009. [Last visited at June 23th 2011].

[7] L. DeMichiel and B. Shannon. JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification. `http://www.jcp.org/en/jsr/detail?id=342`, 2011. Last visited at May 26th 2011.

[8] B. Desmet, J. Vallejos, and P. Costanza. Introducing mixin layers to support the development of context-aware systems. In *3rd European Workshop on Aspects in Software (EWAS 2006)*, August 2006.

[9] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44:29–32, October 2001.

[10] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, January 2004.

[11] S. González, K. Mens, and A. Cádiz. Context-oriented programming with the ambient object system. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[12] Google, Inc. Google App Engine. `http://code.google.com/appengine/`.

[13] Google Inc. Guice. `http://code.google.com/p/google-guice/`.

[14] Google Inc. Implementing multitenancy using namespaces. `http://code.google.com/appengine/docs/java/multitenancy/multitenancy.html`.

[15] Google Inc. Injecting providers. `http://code.google.com/p/google-guice/wiki/InjectingProviders`.

[16] Google Inc. The Memcache Java API. `http://code.google.com/appengine/docs/java/memcache/`.

[17] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *CEC/EEE '07: The 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, pages 551–558, July 2007.

[18] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[19] G. HPI Software Architecture Group. The Context-oriented Programming project. `http://www.hpi.uni-potsdam.de/hirschfeld/projects/cop/`, 2006-2011.

[20] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11), 1990.

[21] C. Osipov, G. Goldszmidt, M. Taylor, and I. Poddar. Develop and deploy multi-tenant web-delivered solutions using ibm middleware: Part 2: Approaches for enabling multi-tenancy. `http://www.ibm.com/developerworks/webservices/library/ws-multitenantpart2/index.html`, May 2009.

[22] SpringSource. Dependency injection and inversion of control. `http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/overview.html#overview-dependency-injection`.

[23] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. Software as a Service: Configuration and customization perspectives. In *SERVICES-2 '08: IEEE Congress on Services Part II*, pages 18–25, Sept. 2008.

[24] L. Tao. Shifting paradigms with the application service provider model. *Computer*, 34(10):32–39, Oct 2001.

[25] E. Truyen, B. Vanhaute, B. N. Jørgensen, W. Joosen, and P. Verbaeten. Dynamic and selective combination of extensions in component-based applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 233–242, Washington, DC, USA, 2001. IEEE Computer Society.

[26] C.-H. Tsai, Y. Ruan, S. Sahu, A. Shaikh, and K. G. Shin. Virtualization-based techniques for enabling multi-tenant management tools. In A. Clemm, L. Z. Granville, and R. Stadler, editors, *DSOM*, volume 4785 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2007.

[27] S. Walraven, E. Truyen, and W. Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *ACM/IFIP/USENIX 12th International Middleware Conference*, volume 7049 of *Lecture Notes in Computer Science*, pages 370–389. IFIP/Springer, December 2011.